#In the name of Allah

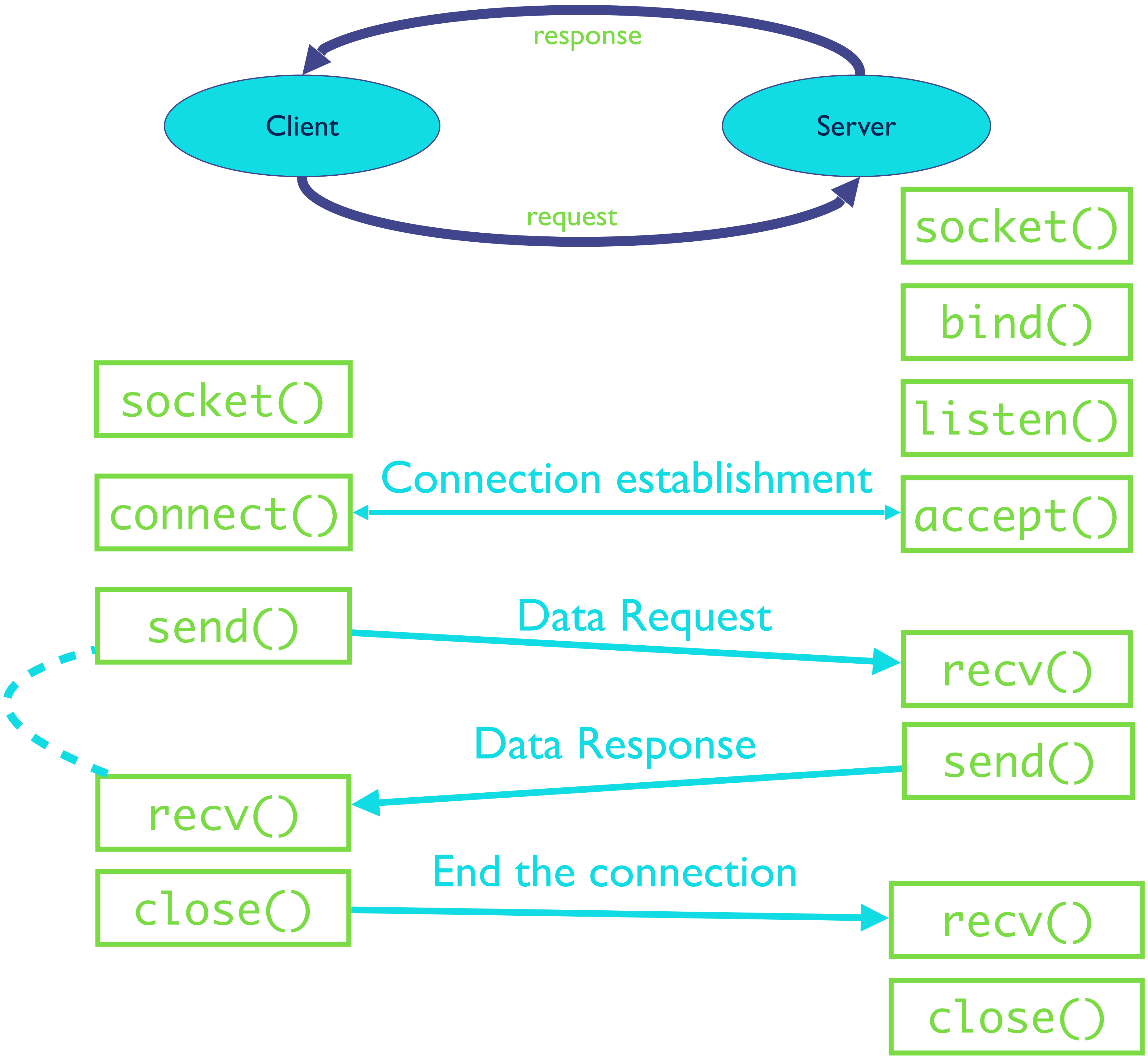
Computer Engineering Department
Sharif University of Technology

CE443- Computer Networks

Socket Programming

Acknowledgments: Lecture slides are from Computer networks course thought by Jennifer Rexford at Princeton University. When slides are obtained from other sources, a reference will be noted on the bottom of that slide.

#Typical Client-Server



#Client Programming

- Create stream socket (socket())
- Connect to server (connect())
- While still connected:
  - send message to server (send())
  - receive (recv()) data from server and process it
- Close TCP connection and Socket (close())

#Client Creating a Socket: socket()

```
int socket(int domain, int type, int protocol)
```

Operation to create a socket

- ✓ Returns a descriptor (or handle) for the socket
- ✓ Originally designed to support any protocol suite

Domain: protocol family

- ✓ PF_INET for the Internet

Type: semantics of the communication

- ✓ SOCK_STREAM: reliable byte stream
- ✓ SOCK_DGRAM: message-oriented service

Protocol: specific protocol

- ✓ UNSPEC: unspecified
- ✓ (PF_INET and SOCK_STREAM already implies TCP)

#Client: Send/Rcv Data and Close

```
int connect(int sockfd, struct sockaddr
*server_address,socketlen_t addrlen)
```

Client contacts the server to establish connection

- ✓Associate the socket with the server address/port
- ✓Acquire a local port number (assigned by the OS)
- ✓Request connection to server, who will hopefully accept

Establishing the connection

- ✓Arguments: socket descriptor, server address, and address
- ✓size
- ✓Returns 0 on success, and -1 if an error occurs

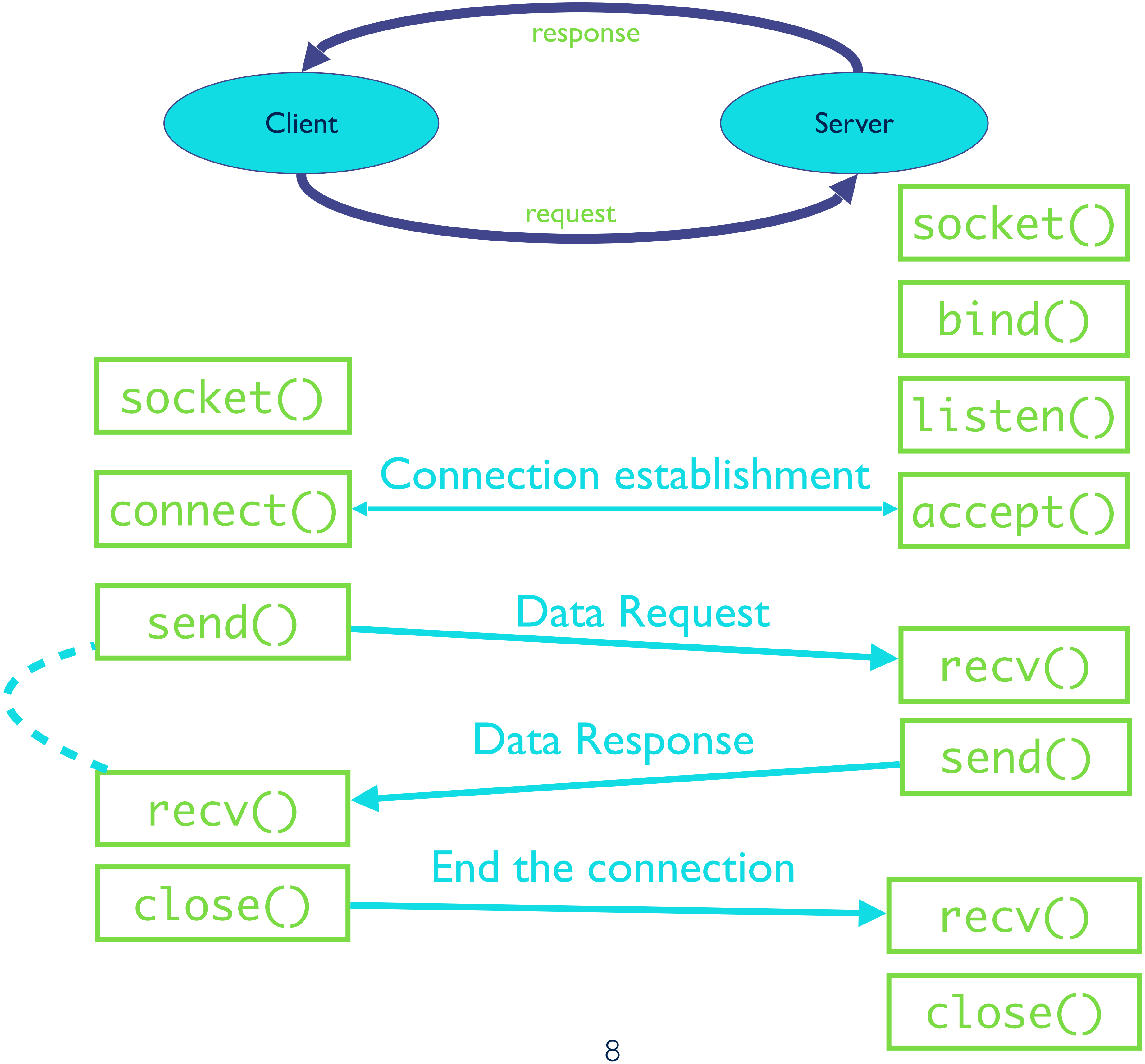
#Programming in Python: Client

```
#!/usr/bin/python
# client.py
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 9999
# connection to hostname on the port.
s.connect((host, port))
# Receive data no more than 1024 bytes
data = s.recv(1024)
s.close()
print("The time got from the server is %s" % tm)
```

#Server Programming: Servers Differ From Clients

- Passive open
    - Prepare to accept connections
    - … but don’t actually establish
    - … until hearing from a client
- Hearing from multiple clients
    - Allowing a backlog of waiting clients
    - ... in case several try to communicate at once
- Create a socket for each client
    - Upon accepting a new client
    - … create a new socket for the communication

#Typical Client-Server



response

Client

Server

request

socket()

bind()

socket()

listen()

connect()

Connection establishment

accept()

send()

Data Request

recv()

Data Response

send()

recv()

close()

End the connection

recv()

close()

#Server Programming: Preparing its Socket

- Create stream socket (socket())
- Bind port to socket (bind()) # local host and port
- Listen for new client (listen()) # How many clients?

#Server Programming: Handle No. of Clients

Many client requests may arrive

- Server cannot handle them all at the same time
- Server could reject the requests, or let them wait
- Define how many connections can be pending: backlog

Wait for clients

- int listen(int sockfd, int backlog)
- Arguments: socket descriptor and acceptable backlog
- Returns a 0 on success, and -1 on error

What if too many clients arrive?

- Some requests don't get through
- The Internet makes no promises…
- And the client can always try again

#Server Programming: Accepting Client Connection

Now all the server can do is wait…

- Waits for connection request to arrive
- Blocking until the request arrives
- And then accepting the new request

Accept a new connection from a client

- int accept(int sockfd, struct sockaddr *addr, socketlen_t
- *addrlen)
- Arguments: socket descriptor, structure that will provide
- client address and port, and length of the structure
- Returns descriptor for a new socket for this connection

#Server Programming: Accepting Client Connection

Serializing requests is inefficient

- Server can process just one request at a time
- All other clients must wait until previous one is done
- May need to time share the server machine

Alternate between servicing different requests

- E.g. use multi-threading
- Or, start a new process to handle each request
- Allow the operating system to share the CPU across processes
- Or, some hybrid of these two approaches

#Client and Server: Cleaning House

Once the connection is open

- Both sides and read and write
- Two unidirectional streams of data
- In practice, client writes first, and server reads
- … then server writes, and client reads, and so on

Closing down the connection

- Either side can close the connection
- … using the close() system call

What about the data still "in flight"

- Data in flight still reaches the other end
- So, server can close() before client finishing reading

#Programming in Python: Server

```
#!/usr/bin/python
#server.py
import socket
server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
# get local machine name
host = socket.gethostname()
port = 9999
# bind to the port
server_socket.bind((host, port))
# queue up to 5 requests
server_socket.listen(5)
while True:
    client_socket,addr = server_socket.accept()
    print "Got a connection from %s" % str(addr)
    my_response = “Hi we are connected!”
    client_socket.send(my_response)
    client_socket.close()
```

socket_family

socket_type